Modified Counting Sort

Ravin Kumar
Department of Computer Science,
Meerut Institute of Engineering and Technology,
Meerut-250005, Uttar Pradesh, India.
ravin.kumar.cs.2013@miet.ac.in

Abstract. There are various sorting methods in the literature which are sequential in nature and have linear time complexity. But these methods are not preferred to use due to large memory requirements in specific cases. Counting sort is one, which lies in this domain. In this paper we have suggested an improvement on the counting sort. Due to this improvement the memory requirement for counting sort is reduced up to a significant level. We have tested this modified counting sort on numerous data sets and the results obtained by these experiments are very much satisfactory. Results shows that this memory requirement is reduced at least 50% than traditional counting sort. So it open up the opportunity of using this modified version in many sorting application.

Keywords: Sorting, Linear sorting, Counting Sort, Modified Counting Sort, Non- Comparison Sorting.

1 Introduction

Sorting is the process of arranging numbers, alphabets or characters in a statistical order (in increasing or decreasing order) or in lexicographical order (alphabetical value like addressee key) [1-3]. These techniques are divided into categories based on their features. Research efforts are made to improve their performance by reducing their complexity. These algorithms are mainly divided into two categories Comparison based sorting and Non-comparison based sorting. Comparison based sorting algorithms involve comparing the values in the array to obtain the sorted sequence. Depending on the type of approach used to perform comparisons, few mainly used algorithms with their complexities are as stated:

1.1 Bubble sort: The algorithm is slow compared to other algorithms. It has a complexity of $O(n^2)$ [4].

1.2 Insertion sort: Insertion sort is a sorting algorithm that works similar to sorting cards in hand. This algorithm also have the complexity of $O(n^2)$ [5].

1.3 Merge sort: This algorithm uses divide and conquer approach [6-7]. It has a complexity of O(n log n).

1.4 Quick sort: Quick Sort is more efficient than the preceding sorting algorithms. It has best case complexity of O(n log n) and worst case complexity of $O(n^2)$ [8].

Non-comparison based sorting algorithms have less complexity because they do not require comparison operation among its elements to obtain the sorted sequence. Counting sort [9] is one classic algorithm used for non-comparison based sorting. It has linear time complexity (i.e. ~ O(n)), but is not preferred for sorting arrays having large numbers because it requires large amount of memory to obtain the sorted sequence. This paper presents a new approach of sorting which leads to a reduction of at least 50 percent of memory usage with a time complexity of ~ O(n).

2 Proposed Algorithm

Traditional Counting Sort has a time complexity of ~ O(n). The amount of memory required to perform sorting using counting sort depends primarily on the largest element present in the input array and less on the total number of elements in the sequence. In a sequence where 'T' is the largest number, It will require an additional array of size 'T + 1' with index ranging from '0' to 'T'. For Example, consider the case where we are sorting the values 1000 and 1 using traditional counting sort. Sorting these values require an additional array of size 1001 as the largest number in the input sequence is 1000.

Modified Counting sort is designed to sort an array in the time complexity of ~ O(n), and also reducing the memory requirement of additional array by at least 50 percent. In the proposed algorithm, the details of parameters and variables are described in table 1.

Table 1. Description of the parameters and variables used in modified counting sort.

Parameter	**Description**
RA	Input array that is to be sorted.
min and **max**	Represents minimum and maximum value respectively.
RB and **RC**	Represents two arrays.
Ref1 and **Ref2**	Represents minimum value in RB and RC respectively.
RS	Represents the sorted array.

Modified_Counting_Sort(RA)

BEGIN

```
min = ∞
max = - ∞
 /*  min and max are calculated from RA
  */
for  i = 0 to Length[ RA ] -1
do
   if  RA [ i ] > max
   then
       max = RA [ i ]
   end if
   if  RA [ i ] < min
   then
       min = RA [ i ]
   end if
end for
div = (min + max) / 2
x=0
y=0
Ref1 = min
Ref2 = ∞
RBmax = - ∞        // RA is divided in two  arrays
for  i = 0 to Length[ RA ] - 1
do
   if  RA [ i ] < div
   then
```

```
        /* Ref1 is subtracted from all the values in  RB, and max value of RB is
           calculated   and   stored  in  RBmax */
        if  RA [ i ] > RBmax
        then
             RBmax = RA [ i ]
        end if
        RB [ x ] = RA [ i ] - Ref1
        x++
    else
        if  RA [ i ] < Ref2
        then
              Ref2=RA [ i ]    /* Ref2 of RC is calculated.  */
        end if
        RC [ y ]= RA [ i ]
        y++
    end if
end for
for  i = 0 to Length[ RC ] -1
do
    RC [ i ] = RC [ i ] - Ref2
end for
/* Now applying the  New_Counting _Sort  on   both   RB and RC sub arrays. */
New_Counting_Sort(RB, RS, RBmax - Ref1, Ref1, 0)
New_Counting_Sort(RC, RS, max - Ref2, Ref2, Length[RB] )
END
```

New_Counting_Sort(A, B, k, Ref, AddLen)

A is the input array, B is the sorted array, and k is the maximum number which is present in A array.

```
BEGIN
 for  i = 0 to k
 do
    C[i] = 0
 end for
 for  j = 0 to Length[A] -1
 do
    C [ A [ j ] ] = C [ A [ j ] ] + 1
 end for
 for  i = 1  to k
 do
```

```
    C [ i ]  = C [ i ]  + C [ i – 1]
 end for
 for  j = Length[A] - 1 down to 0
 do
    B [ C [ A [ j ] ]  + AddLen - 1] = A [ j ] + Ref
    C [ A[ j ] ] = C [ A [ j ] ] -1
 end for
```
END

The time complexity of the modified counting sort is still O(n) and it reduces the amount of memory required to perform sorting operation.

3 Working Demonstration

Consider the following input data set on which we have to perform sorting operation using the proposed algorithm.

20	33	23	60	52	210	240	200	130	42

On applying the proposed algorithm over this data. We get min=20 , max=240 and div=130. Values less than div are:

20	33	23	60	52	42

Similarly, values greater than or equal to div are:

210	240	200	130

From the algorithm, we get Ref1 = 20 and Ref2 = 130.

Now after subtracting Ref1 from first array, store the obtained data in RB. Similarly, after subtracting Ref2 from second array, store the obtained data in RC.

After performing above operation, elements in RB becomes :

0	13	3	40	32	22

Similarly, elements present in RC become:

80	110	70	0

Now our modified counting sort uses New_Counting_Sort on RB, which sorts RB and then adds the Ref1 value to the result of sorted RB :

0+20	3+20	13+20	22+20	32+20	40+20

Obtained result in RB :

20	23	33	42	52	60

And the size of C[] for doing this operation was 41 (i.e. from 0 to 40).

Now our modified counting sort uses New_Counting_Sort on RC, which sorts RC and then adds the Ref1 value to the result of sorted RC:

0+130	70+130	80+130	110+130

Obtained result in RC:

130	200	210	240

Now Final result of sorting is obtained in RS, and the sorted sequence becomes :

20	23	33	42	52	60	130	200	210	240

Total memory required by additional C[ ] is 111.(i.e. 0 to 110),

while on applying traditional counting sort, it would require an additional array of size 241, indexed from 0 to 240 to sort this input sequence.

4 Result And Discussion

Modified Counting sort is tested on various input sets, and a comparison is done with the traditional counting sort algorithm. Since both algorithms have time complexity of O(n), comparison is done on the basis of amount of memory required as shown in table 2.

Table 2. Sample Data to demonstrate the comparison between traditional counting sort and modified counting sort.

Input Elements	**Memory of C[ ] in Counting sort (i.e. length of C [ ])**	**Memory of C [ ] in Modified_Counting_Sort (i.e. length of C [ ])**
1,2,3,250,251,252	253	3
18, 21, 19, 23, 407, 401, 402, 400	408	8
1000,1001,1002,1003, 1004, 1005	1006	3
900, 800, 2, 1, 700, 300, 4, 7, 9, 12, 27	901	300
9,8,11,10,1001,1700,1500,2000	2001	1000
20, 100, 1, 0	101	21
1000,999,998,997... ,3,2,1	1001	500

To understand the effectiveness of the proposed algorithm the comparison using the above data set can be displayed graphically as shown in figure 1.

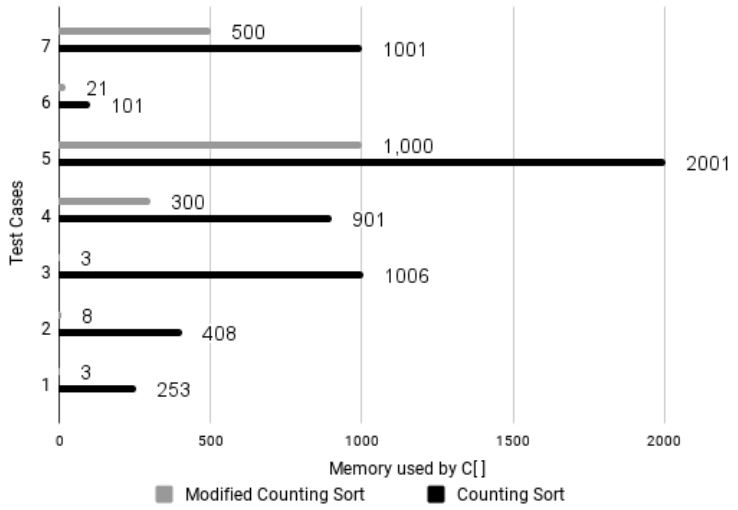

Fig. 1. Representation of memory used by each test data sequence in both traditional and modified counting sorts.

From the data shown in Figure 1, it is clear that in worst case, the memory used by the modified counting is half the memory used by the traditional counting sort algorithm. The time complexity of both the algorithms remains ~ O(n). This can also be graphically represented as shown in figure 2.

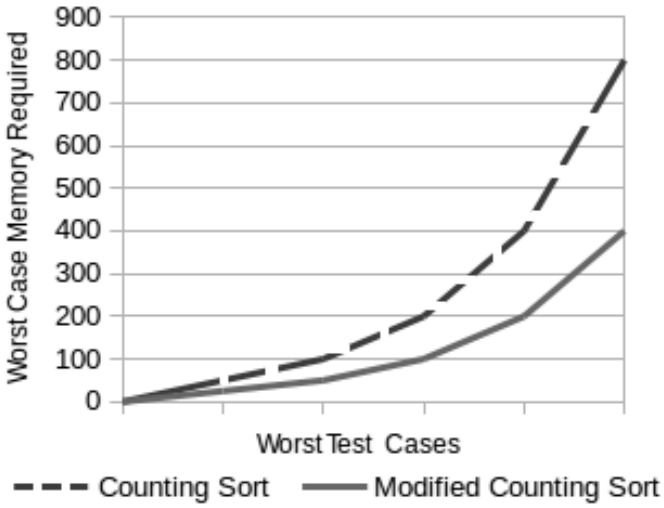

Fig. 2. Representation of comparison of worst case memory requirement by traditional and modified counting sorts.

5 Conclusion

In this paper, an advanced method is introduced which reduces the memory requirements of the classic Counting sort algorithm by at least 50 percent. Modified Counting Sort can be used for improving performance of various existing sorting algorithms like- radix sort [10], which requires an additional stable sorting algorithm to perform sorting. Using Modified counting sort, It is possible to improve the current sorting applications like Searching a value in a large array, Sorting in large databases and TCP/IP packet sorting[11].

References

1. I. Flores, “Analysis of Internal Computer Sorting”, ACM, vol.7, no. 4, (1960), pp. 389-409.

2. G. Franceschini and V. Geffert, “An In-Place Sorting with O(n log n) Comparisons and O(n) Moves”, Proceedings of 44th Annual IEEE Symposium on Foundations of Computer Science, (2003), pp. 242-250.
3. D. Knuth, “The Art of Computer programming Sorting and Searching”, 2nd edition, Addison-Wesley,vol. 3,(1998).
4. Oyelami Olufemi Moses, “Improving the Performance of Bubble Sort Using a Modified Diminishing Increment Sorting,” Scientific Research and Essay Vol.4 (8), pp. 740 -744 (2009).
5. Rupesh Srivastava, Tarun Tiwari, and Sweetes Sing, “Bidirectional Expansion—Insertion Algorithm for Sorting,” Second International Conference on Emerging Trends in Engineering and Technology, ICETET-09. (2009).
6. Radu Rudina and Martin Rinard, “Automatic Parallelization of Divide and Conquer Algorithm”, Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.
7. Dean C., “A Simple Expected Running Time Analysis for Randomized Divide and Conquer Algorithms,” Computer Journal of Discrete Applied Mathematics, vol. 154, no. 1, pp. 15, 2006.
8. Friend E., “Sorting on Electronic Computer Systems,” Computer Journal of ACM, vol. 3, no. 3, pp. 134168, 1956.
9. M Rajasekhara Babu, M Khalid, Sachin Soni, Sunil Chowdari Babu, Mahesh (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (5),2011, 2284-2287.
10. A. Andersson and S. Nilsson. A new efficient radix sort.In Proc. 35th Annual IEEE Symp. on Foundations of Computer Science, pp. 714-721, 1994.
11. C. Meinel and H. Sack, Internetworking, X.media.publishing, DOI: 10.1007/978-3-642-35392-5_2, Ó Springer-Verlag Berlin Heidelberg 2013.